Oracle® Solaris 11.4 Programming Interfaces Guide

Part No: E61059
August 2018

♦ ♦ ♦ CHAPTER 6

Interprocess Communication

This chapter is for programmers who develop multiprocess applications.

SunOS 5.11 and compatible operating systems support different mechanisms for concurrent processes to exchange data and synchronize execution. All of these mechanisms, except mapped memory, are introduced in this chapter.

- Pipes (anonymous data queues) are described in "Pipes Between Processes" on page 105.
- Named pipes (data queues with file names.) are described in "Named Pipes" on page 107
.
- System V message queues, semaphores, and shared memory are described in "System V IPC" on page 110.
- POSIX message queues, semaphores, and shared memory are described in "POSIX Interprocess Communication" on page 108.
- Interprocess communication using sockets are described in "Sockets Overview" on page 107.
- Mapped memory and files are described in "Memory Management Interfaces" on page 15.
- Doors (a mechanism for secure control transfer) are described in "Doors Overview" on page 107.

Pipes Between Processes

A pipe between two processes is a pair of files that is created in a parent process. The pipe connects the resulting processes when the parent process forks. A pipe does not exist in any file name space, so it is referred as anonymous. A pipe connects only two processes. A single pipe also connects multiple child processes to each other and their related parent.

A pipe is created in the process that becomes the parent by a call to `pipe`. The call returns two file descriptors in the array passed to it. After forking, both processes read from `p[0]` and write to `p[1]`. The processes read from and write to a circular buffer that is managed for them. For more information, see the `pipe(2)` man page.

Calling `fork` duplicates the per-process open file table. Each process has two readers and two writers. Closing the extra readers and writers enables the proper functioning of the pipe. For example, if the end of a reader is left open by the same process for writing, no end-of-file indication is returned. For more information, see the `fork(2)` man pages.

The following code shows pipe creation, a fork, and clearing the duplicate pipe ends.

```
#include <stdio.h>
#include <unistd.h>
...
        int p[2];
...
        if (pipe(p) == -1) exit(1);
        switch( fork() )
        {
                case 0:                  /* in child */
                        close( p[0] );
                        dup2( p[1], 1);
                        close P[1] );
                        exec( ... );
                        exit(1);
                default:                 /* in parent */
                        close( p[1] );
                        dup2( P[0], 0 );
                        close( p[0] );
                        break;
        }
        ...
```

The following table shows the results of reads from a pipe and writes to a pipe, under certain conditions.

TABLE 6 Read/Write Results in a Pipe

Attempt	Conditions	Result
read	Empty pipe, writer attached	Read blocked
write	Full pipe, reader attached	Write blocked
read	Empty pipe, no writer attached	EOF returned
write	No reader	`SIGPIPE`

Blocking can be prevented by calling `fcntl` on the descriptor to set `FNDELAY`. This causes an error return (-1) from the I/O call with `errno` set to `EWOULDBLOCK`. For more information, see the `fcntl(2)` man page.

Named Pipes

Named pipes function much like pipes, but are created as named entities in a file system. This enables the pipe to be opened by all processes with no requirement that they be related by forking. A named pipe is created by a call to `mknod`. Any process with appropriate permission can then read or write to a named pipe. For more information, see the `mknod(2)` man page.

In the `open` call, the process opening the pipe blocks until another process also opens the pipe.

To open a named pipe without blocking, the `open` call joins the `O_NDELAY` mask (found in `sys/fcntl.h`) with the selected file mode mask using the Boolean `or` operation on the call to `open`. If no other process is connected to the pipe when `open` is called, -1 is returned with `errno` set to `EWOULDBLOCK`. For more information, see the `open(2)` man page.

Sockets Overview

Sockets provide point-to-point, two-way communication between two processes. Sockets are a basic component of interprocess and inter-system communication. A socket is an endpoint of communication to which a name can be bound. It has a type and one or more associated processes.

Sockets exist in communication domains. A socket domain is an abstraction that provides an addressing structure and a set of protocols. Sockets connect only with sockets in the same domain. Twenty three socket domains are identified (see `sys/socket.h`), of which only the UNIX and Internet domains are normally used in Oracle Solaris OS and compatible operating systems.

You can use sockets to communicate between processes on a single system, like other forms of IPC. The UNIX domain (`AF_UNIX`) provides a socket address space on a single system. UNIX domain sockets are named with UNIX paths. UNIX domain sockets are further described in Appendix A, “UNIX Domain Sockets”. Sockets can also be used to communicate between processes on different systems. The socket address space between connected systems is called the Internet domain (`AF_INET`). Internet domain communication uses the TCP/IP internet protocol suite. Internet domain sockets are described in Chapter 7, “Socket Interfaces”.

Doors Overview

Doors are a fast light-weight RPC mechanism for secure control transfer between processes on the same machine. A door is created when a process known as the door server calls door_create(3DOOR) with a server function and receives a file descriptor. The file descriptor can be passed to other processes or attached to the file system using fattach(3C). A client process, which has the file descriptor, can then invoke the door process by calling door_call(3DOOR). The client can also pass data and descriptors including other door descriptors. As a result of the call to door_call(), the client thread blocks and a thread in the door server wakes up and starts running the server function. When the server function is completed, the function calls door_return(3DOOR) to pass optional data and descriptors back to the client. door_return() also switches control back to the client; the server thread gets blocked in the kernel and does not return from the door_return call.

Doors are described in the doors library libdoor(3LIB).

POSIX Interprocess Communication

POSIX interprocess communication (IPC) is a variation of System V interprocess communication. Like System V objects, POSIX IPC objects have read and write, but not execute, permissions for the owner, the owner's group, and for others. There is no way for the owner of a POSIX IPC object to assign a different owner. POSIX IPC includes the following features:

- Messages allow processes to send formatted data streams to arbitrary processes.
- Semaphores allow processes to synchronize execution.
- Shared memory allows processes to share parts of their virtual address space.

Unlike the System V IPC interfaces, the POSIX IPC interfaces are all multithread safe.

POSIX Messages

The POSIX message queue interfaces are listed in the following table.

TABLE 7 POSIX Message Queue Interfaces

Interface Name	Purpose
mq_open	Connects to and optionally creates a named message queue
mq_close	Ends the connection to an open message queue
mq_unlink	Ends the connection to an open message queue and causes the queue to be removed when the last process closes it
mq_send	Places a message in the queue

Interface Name	Purpose
`mq_receive`	Receives (removes) the oldest, highest priority message from the queue
`mq_notify`	Notifies a process or thread that a message is available in the queue
`mq_setattr`	Set or get message queue attributes

POSIX Semaphores

POSIX semaphores are much lighter weight than are System V semaphores. A POSIX semaphore structure defines a single semaphore, not an array of up to 25 semaphores.

The POSIX semaphore interfaces are shown below:

`sem_open`	Connects to, and optionally creates, a named semaphore
`sem_init`	Initializes a semaphore structure (internal to the calling program, not a named semaphore)
`sem_close`	Ends the connection to an open semaphore
`sem_unlink`	Ends the connection to an open semaphore and causes the semaphore to be removed when the last process closes it
`sem_destroy`	Initializes a semaphore structure (internal to the calling program, not a named semaphore)
`sem_getvalue`	Copies the value of the semaphore into the specified integer
`sem_wait`	Blocks while the semaphore is held by other processes or returns an error if the semaphore is held by another process

POSIX Shared Memory

POSIX shared memory is actually a variation of mapped memory (see "Creating and Using Mappings" on page 15). The major differences are:

- You use `shm_open` to open the shared memory object instead of calling `open`.
- You use `shm_unlink` to close and delete the object instead of calling `close` which does not remove the object.

The options in `shm_open` substantially fewer than the number of options provided in `open`.

System V IPC

SunOS 5.11 and compatible operating systems also provide the System V inter process communication (IPC) package. System V IPC has effectively been replaced by POSIX IPC, but is maintained to support older applications.

For more information about the Sysytem V IPC, see the `ipcrm`(1), `ipcs`(1), `Intro`(2), `msgctl`(2), `msgget`(2), `msgrcv`(2), `msgsnd`(2), `semget`(2), `semctl`(2), `semop`(2), `shmget`(2), `shmctl`(2), `shmop`(2), and `ftok`(3C) man pages.

Permissions for Messages, Semaphores, and Shared Memory

Messages, semaphores, and shared memory have read and write permissions, but no execute permission, for the owner, group, and others, which is similar to ordinary files. Like files, the creating process identifies the default owner. Unlike files, the creating process can assign ownership of the facility to another user or revoke an ownership assignment.

IPC Interfaces, Key Arguments, and Creation Flags

Processes requesting access to an IPC facility must be able to identify the facility. To identify the facility to which the process requests access, interfaces that initialize or provide access to an IPC facility use a `key_t` *key* argument. The *key* is an arbitrary value or one that can be derived from a common seed at runtime. One way to derive such a key is by using `ftok`, which converts a file name to a key value that is unique within the system. For more information, see the `ftok`(3C) man page.

Interfaces that initialize or get access to messages, semaphores, or shared memory return an ID number of type `int`. IPC Interfaces that perform read, write, and control operations use this ID.

If the key argument is specified as `IPC_PRIVATE`, the call initializes a new instance of an IPC facility that is private to the creating process.

When the `IPC_CREAT` flag is supplied in the flags argument appropriate to the call, the interface tries to create the facility if it does not exist already.

When called with both the`IPC_CREAT` and `IPC_EXCL` flags, the interface fails if the facility already exists. This behavior can be useful when more than one process might attempt to initialize the facility. One such case might involve several server processes having access to the same facility. If they all attempt to create the facility with `IPC_EXCL` in effect, only the first attempt succeeds.

If neither of these flags is given and the facility already exists, the interfaces return the ID of the facility to get access. If `IPC_CREAT` is omitted and the facility is not already initialized, the calls fail.

Using logical (bitwise) `OR`, `IPC_CREAT` and `IPC_EXCL` are combined with the octal permission modes to form the flags argument. For example, the statement below initializes a new message queue if the queue does not exist:

```
msqid = msgget(ftok("/tmp", 'A'), (IPC_CREAT | IPC_EXCL | 0400));
```

The first argument evaluates to a key (`'A'`) based on the string (`"/tmp"`). The second argument evaluates to the combined permissions and control flags.

System V Messages

Before a process can send or receive a message, you must initialize the queue through `msgget` . The owner or creator of a queue can change its ownership or permissions using `msgctl`. Any process with permission can use `msgctl` for control operations. For more information, see the `msgget(2)` and `msgctl(2)` man pages.

IPC messaging enables processes to send and receive messages and queue messages for processing in an arbitrary order. Unlike the file byte-stream data flow of pipes, each IPC message has an explicit length.

Messages can be assigned a specific type. A server process can direct message traffic between clients on its queue by using the client process PID as the message type. For single-message transactions, multiple server processes can work in parallel on transactions sent to a shared message queue.

Operations to send and receive messages are performed by `msgsnd` and `msgrcv`, respectively. When a message is sent, its text is copied to the message queue. `msgsnd` and `msgrcv` can be performed as either blocking or non-blocking operations. For more information, see the `msgsnd(2)` and `msgrcv(2)` man pages.

A blocked message operation remains suspended until one of the following three conditions occurs:

- The call succeeds

- The process receives a signal
- The queue is removed

Initializing a Message Queue

`msgget` initializes a new message queue. It can also return the message queue ID (`msqid`) of the queue corresponding to the key argument. The value passed as the `msgflg` argument must be an octal integer with settings for the queue's permissions and control flags.

The `MSGMNI` kernel configuration option determines the maximum number of unique message queues that the kernel supports. `msgget` fails when this limit is exceeded. For more information, see the `msgget(2)` man page.

The following code illustrates `msgget`.

```
#include <sys/ipc.h>
#include <sys/msg.h>
...
        key_t    key;         /* key to be passed to msgget() */
        int      msgflg,      /* msgflg to be passed to msgget() */
                 msqid;       /* return value from msgget() */
        ...
        key = ...
        msgflg = ...
        if ((msqid = msgget(key, msgflg)) == -1)
        {
               perror("msgget: msgget failed");
               exit(1);
        } else
               (void) fprintf(stderr, "msgget succeeded");
...
```

Controlling Message Queues

`msgctl` alters the permissions and other characteristics of a message queue. The `msqid` argument must be the ID of an existing message queue. The `cmd` argument is one of the following:

`IPC_STAT` Place information about the status of the queue in the data structure pointed to by `buf`. The process must have read permission for this call to succeed.

`IPC_SET` Set the owner's user and group ID, the permissions, and the size (in number of bytes) of the message queue. A process must have the effective user ID of the owner, creator, or superuser for this call to succeed.

IPC_RMID — Remove the message queue specified by the msqid argument.

The following code illustrates msgctl with all its flags.

```
#include                        <sys/types.h>
#include                        <sys/ipc.h>
#include                        <sys/msg.h>
        ...
        if (msgctl(msqid, IPC_STAT, &buf) == -1)  {
                perror("msgctl: msgctl failed");
                exit(1);
        }
        ...
        if (msgctl(msqid, IPC_SET, &buf) == –1) {
                perror("msgctl: msgctl failed");
                exit(1);
        }
...
```

Sending and Receiving Messages

msgsnd and msgrcv send and receive messages, respectively. The msqid argument must be the ID of an existing message queue. The msgp argument is a pointer to a structure that contains the type of the message and its text. The msgsz argument specifies the length of the message in bytes. The msgflg argument passes various control flags. For more information, see the msgsnd(2) and msgrcv(2) man pages.

The following code illustrates msgsnd and msgrcv.

```
#include                     <sys/types.h>
#include                     <sys/ipc.h>
#include                     <sys/msg.h>
...
        int               msgflg;        /* message flags for the operation */
        struct msgbuf     *msgp;         /* pointer to the message buffer */
        size_t            msgsz;         /* message size */
        size_t            maxmsgsize;    /* maximum message size */
        long              msgtyp;        /* desired message type */
        int               msqid          /* message queue ID to be used */
        ...
        msgp = malloc(sizeof(struct msgbuf) – sizeof (msgp–>mtext)
                          + maxmsgsz);
        if (msgp == NULL) {
```

```
                (void) fprintf(stderr, "msgop: %s %ld byte messages.\n",
                           "could not allocate message buffer for", maxmsgsz);
                exit(1);
                ...
                msgsz = ...
                msgflg = ...
                if (msgsnd(msqid, msgp, msgsz, msgflg) == –1)
                           perror("msgop: msgsnd failed");
                ...
                msgsz = ...
                msgtyp = first_on_queue;
                msgflg = ...
                if (rtrn = msgrcv(msqid, msgp, msgsz, msgtyp, msgflg) == –1)
                           perror("msgop: msgrcv failed");
...
```

System V Semaphores

Semaphores enable processes to query or alter status information. They are used to monitor and control the availability of system resources such as shared memory segments. Semaphores can be operated on as individual units or as elements in a set.

Because System V IPC semaphores can be in a large array, they are extremely heavy weight. Much lighter-weight semaphores are available in the threads library. Also, POSIX semaphores are the most current implementation of System V semaphores (see "POSIX Semaphores" on page 109). Threads library semaphores must be used with mapped memory. For more information, see "Memory Management Interfaces" on page 15.

A semaphore set consists of a control structure and an array of individual semaphores. A set of semaphores can contain up to 25 elements. The semaphore set must be initialized using `semget`. The semaphore creator can change its ownership or permissions using `semctl`. Any process with permission can use `semctl` to do control operations. For more information, see the `semget(2)` and `semctl(2)` man pages.

Semaphore operations are performed by `semop`. This interface takes a pointer to an array of semaphore operation structures. Each structure in the array contains data about an operation to perform on a semaphore. Any process with read permission can test whether a semaphore has a zero value. Operations to increment or decrement a semaphore require write permission. For more information, see the `semop(2)` man page.

When an operation fails, none of the semaphores are altered. The process blocks unless the `IPC_NOWAIT` flag is set, and remains blocked until:

- The semaphore operations can all finish, so the call succeeds.

- The process receives a signal.
- The semaphore set is removed.

Only one process at a time can update a semaphore. Simultaneous requests by different processes are performed in an arbitrary order. When an array of operations is given by a `semop` call, no updates are done until all operations on the array can finish successfully.

If a process with exclusive use of a semaphore terminates abnormally and fails to undo the operation or free the semaphore, the semaphore stays locked in memory in the state the process left it. To prevent this occurrence, the `SEM_UNDO` control flag makes `semop` allocate an undo structure for each semaphore operation, which contains the operation that returns the semaphore to its previous state. If the process dies, the system applies the operations in the undo structures. This prevents an aborted process from leaving a semaphore set in an inconsistent state. For more information, see the `semop(2)` man page.

If processes share access to a resource controlled by a semaphore, operations on the semaphore should not be made with `SEM_UNDO` in effect. If the process that currently has control of the resource terminates abnormally, the resource is presumed to be inconsistent. Another process must be able to recognize this to restore the resource to a consistent state.

When performing a semaphore operation with `SEM_UNDO` in effect, you must also have `SEM_UNDO` in effect for the call that performs the reversing operation. When the process runs normally, the reversing operation updates the undo structure with a complementary value. This ensures that, unless the process is aborted, the values applied to the undo structure are canceled to zero. When the undo structure reaches zero, it is removed.

Using `SEM_UNDO` inconsistently can lead to memory leaks because allocated undo structures might not be freed until the system is rebooted.

Initializing a Semaphore Set

`semget` initializes or gains access to a semaphore. When the call succeeds, it returns the semaphore ID (`semid`). The key argument is a value associated with the semaphore ID. The `nsems` argument specifies the number of elements in a semaphore array. The call fails when `nsems` is greater than the number of elements in an existing array. When the correct count is not known, supplying 0 for this argument ensures that it will succeed. The `semflg` argument specifies the initial access permissions and creation control flags. For more information, see the `semget(2)` man page.

The `SEMMNI` system configuration option determines the maximum number of semaphore arrays allowed. The `SEMMNS` option determines the maximum possible number of individual semaphores across all semaphore sets. Because of fragmentation between semaphore sets, allocating all available semaphores might not be possible.

The following code illustrates `semget`.

```
#include                        <sys/types.h>
#include                        <sys/ipc.h>
#include                        <sys/sem.h>
...
         key_t    key;       /* key to pass to semget() */
         int      semflg;    /* semflg to pass to semget() */
         int      nsems;     /* nsems to pass to semget() */
         int      semid;     /* return value from semget() */
         ...
         key = ...
         nsems = ...
         semflg = ...
         ...
         if ((semid = semget(key, nsems, semflg)) == –1) {
                 perror("semget: semget failed");
                 exit(1);
         } else
                 exit(0);
...
```

Controlling Semaphores

`semctl` changes permissions and other characteristics of a semaphore set. It must be called with a valid semaphore ID. The `semnum` value selects a semaphore within an array by its index. The *cmd* argument is one of the following control flags.

`GETVAL` Return the value of a single semaphore.

`SETVAL` Set the value of a single semaphore. In this case, `arg` is taken as `arg.val`, an `int`.

`GETPID` Return the `PID` of the process that performed the last operation on the semaphore or array.

`GETNCNT` Return the number of processes waiting for the value of a semaphore to increase.

`GETZCNT` Return the number of processes waiting for the value of a particular semaphore to reach zero.

`GETALL` Return the values for all semaphores in a set. In this case, `arg` is taken as `arg.array`, a pointer to an array of `unsigned short` values.

`SETALL`	Set values for all semaphores in a set. In this case, `arg` is taken as `arg.array`, a pointer to an array of `unsigned short` values.
`IPC_STAT`	Return the status information from the control structure for the semaphore set and place it in the data structure pointed to by `arg.buf`, a pointer to a buffer of type `semid_ds`.
`IPC_SET`	Set the effective user and group identification and permissions. In this case, `arg` is taken as `arg.buf`.
`IPC_RMID`	Remove the specified semaphore set.

A process must have a user identification of the owner, the creator, or the superuser to perform an `IPC_SET` or `IPC_RMID` command. For other control commands read and write permission is required.

The following code illustrates `semctl`.

```
#include                     <sys/types.h>
#include                     <sys/ipc.h>
#include                     <sys/sem.h>
...
        register int         i;
...
        i = semctl(semid, semnum, cmd, arg);
        if (i == -1) {
               perror("semctl: semctl failed");
               exit(1);
...
```

Semaphore Operations

`semop` performs operations on a semaphore set. The `semid` argument is the semaphore ID returned by a previous `semget(2)` call. The `sops` argument is a pointer to an array of structures, each containing the following information about a semaphore operation:

- The semaphore number
- The operation to be performed
- Control flags, if any

The `sembuf` structure specifies a semaphore operation, as defined in `sys/sem.h`. The `nsops` argument specifies the length of the array, the maximum size of which is determined by the *SEMOPM* configuration option. This option determines the maximum number of operations allowed by a single `semop` call, and is set to 10 by default.

The operation to be performed is determined as follows:

- Positive integer increments the semaphore value by the specified amount.
- Negative integer decrements the semaphore value by the specified amount. An attempt to set a semaphore to a value less than zero fails or blocks, depending on whether IPC_NOWAIT is in effect.
- Value zero means to wait for the semaphore value to reach zero.

The two control flags that can be used with semop are IPC_NOWAIT and SEM_UNDO.

IPC_NOWAIT
: Can be set for any operations in the array. Makes the interface return without changing any semaphore value if it cannot perform any of the operations for which IPC_NOWAIT is set. The interface fails if it tries to decrement a semaphore more than its current value, or tests a nonzero semaphore to be equal to zero.

SEM_UNDO
: Allows individual operations in the array to be undone when the process exits.

The following code illustrates semop.

```
#include                                <sys/types.h>
#include                                <sys/ipc.h>
#include                                <sys/sem.h>
...
         int              i;            /* work area */
         int              nsops;        /* number of operations to do */
         int              semid;        /* semid of semaphore set */
         struct sembuf    *sops;        /* ptr to operations to perform */
         ...
         if ((i = semop(semid, sops, nsops)) == –1) {
                 perror("semop: semop failed");
         } else
                 (void) fprintf(stderr, "semop: returned %d\n", i);
...
```

For more information, see the semop(2) man page,

System V Shared Memory

In the SunOS 5.11 operating system, the efficient way to implement shared memory applications is to rely on mmap and on the system's native virtual memory facility. For more information, see Chapter 1, “Memory and CPU Management” and the mmap(2) man page.

The SunOS 5.11 platform also supports System V shared memory, which is a less efficient way to enable the attachment of a segment of physical memory to the virtual address spaces of multiple processes. When write access is allowed for more than one process, an outside protocol or mechanism, such as a semaphore, can be used to prevent inconsistencies and collisions.

A process creates a shared memory segment using `shmget`. This call is also used to get the ID of an existing shared segment. The creating process sets the permissions and the size in bytes for the segment.

The original owner of a shared memory segment can assign ownership to another user with `shmctl`. The owner can also revoke this assignment. Other processes with proper permission can perform various control functions on the shared memory segment using `shmctl`.

Once created, you can attach a shared segment to a process address space using `shmat`. You can detach it using `shmdt`. The attaching process must have the appropriate permissions for `shmat`. Once attached, the process can read or write to the segment, as allowed by the permission requested in the attach operation. A shared segment can be attached multiple times by the same process.

A shared memory segment is described by a control structure with a unique ID that points to an area of physical memory. The identifier of the segment is called the `shmid`. You can find the structure definition for the shared memory segment control in `sys/shm.h`.

For more information, see the `shmget(2)`, `shmctl(2)`, `shmat(2)`, and `shmdt(2)` man pages.

Accessing a Shared Memory Segment

`shmget` is used to obtain access to a shared memory segment. When the call succeeds, it returns the shared memory segment ID (*shmid*). The following code illustrates `shmget`.

```
#include                    <sys/types.h>
#include                    <sys/ipc.h>
#include                    <sys/shm.h>
...
        key_t     key;       /* key to be passed to shmget() */
        int       shmflg;    /* shmflg to be passed to shmget() */
        int       shmid;     /* return value from shmget() */
        size_t    size;      /* size to be passed to shmget() */
        ...
        key = ...
        size = ...
        shmflg) = ...
        if ((shmid = shmget (key, size, shmflg)) == –1) {
               perror("shmget: shmget failed");
```

```
                exit(1);
        } else {
                (void) fprintf(stderr,
                              "shmget: shmget returned %d\n", shmid);
                exit(0);
        }
...
```

Controlling a Shared Memory Segment

`shmctl` is used to alter the permissions and other characteristics of a shared memory segment. The `cmd` argument is one of following control commands.

`SHM_LOCK` — Lock the specified shared memory segment in memory. The process must have the effective ID of superuser to perform this command.

`SHM_UNLOCK` — Unlock the shared memory segment. The process must have the effective ID of superuser to perform this command.

`IPC_STAT` — Return the status information contained in the control structure and place it in the buffer pointed to by `buf`. The process must have read permission on the segment to perform this command.

`IPC_SET` — Set the effective user and group identification and access permissions. The process must have an effective ID of owner, creator or superuser to perform this command.

`IPC_RMID` — Remove the shared memory segment. The process must have an effective ID of owner, creator, or superuser to perform this command.

The following code illustrates `shmctl`.

```
#include                    <sys/types.h>
#include                    <sys/ipc.h>
#include                    <sys/shm.h>
...
int     cmd;                /* command code for shmctl() */
int     shmid;              /* segment ID */
struct  shmid_ds  shmid_ds; /* shared memory data structure to hold results */
        ...
        shmid = ...
```

```
        cmd = ...
        if ((rtrn = shmctl(shmid, cmd, shmid_ds)) == –1) {
                perror("shmctl: shmctl failed");
                exit(1);
...
```

Attaching and Detaching a Shared Memory Segment

`shmat()` and `shmdt()` functions are used to attach and detach shared memory segments. `shmat` returns a pointer to the head of the shared segment. `shmdt()` detaches the shared memory segment located at the address indicated by *shmaddr*. For more information, see the `shmop`(2), `shmat`(2), and `shmdt`(2) man pages.

The following code illustrates calls to `shmat()` and `shmdt`.

```
#include              <sys/types.h>
#include              <sys/ipc.h>
#include              <sys/shm.h>

static struct state { /* Internal record of attached segments. */
        int           shmid;        /* shmid of attached segment */
        char          *shmaddr;     /* attach point */
        int           shmflg;       /* flags used on attach */
} ap[MAXnap];                       /* State of current attached segments. */
int     nap;                        /* Number of currently attached segments. */
...
char    *addr;                      /* address work variable */
register int          i;            /* work area */
register struct state *p;           /* ptr to current state entry */
...
        p = &ap[nap++];
        p–>shmid = ...
        p–>shmaddr = ...
        p–>shmflg = ...
        p–>shmaddr = shmat(p->shmid, p->shmaddr, p->shmflg);
        if(p–>shmaddr == (char *)-1) {
                perror("shmat failed");
                nap–-;
        } else
                (void) fprintf(stderr, "shmop: shmat returned %p\n",
                                    p–>shmaddr);
        ...
        i = shmdt(addr);
        if(i == –1) {
                 perror("shmdt failed");
        } else {
```

```
                        (void) fprintf(stderr, "shmop: shmdt returned %d\n", i);
                        for (p = ap, i = nap; i—-; p++) {
                                if (p—>shmaddr == addr) *p = ap[—-nap];
                        }
                }
        ...
```